Unreliable Heterogeneous Workers in a pool-based evolutionary algorithm

–

No Institute Given

Abstract. In this paper the effect of node unavailability in algorithms using EvoSpace, a pool-based evolutionary algorithm, is assessed. EvoSpace is a framework for developing evolutionary algorithms (EAs) using heterogeneous and unreliable resources. It is based on Linda's tuple space coordination model. The core elements of EvoSpace are a central repository for the evolving population and remote clients, here called EvoWorkers, which pull random samples of the population to perform on them the basic evolutionary processes (selection, variation and survival), once the work is done, the modified sample is pushed back to the central population. To address the problem of unreliable EvoWorkers, EvoSpace uses a simple re-insertion algorithm using copies of samples stored in a global queue which also prevents the starvation of the population pool. Using a benchmark problem from the P-Peaks problem generator we have compared two approaches: (i) the re-insertion of previous individuals at the cost of keeping copies of each sample, and a common approach of other pool based EAs, (ii) inserting randomly generated individuals. We found that EvoSpace is fault tolerant to highly unreliable resources and also that the re-insertion algorithm is only needed when the population is near the point of starvation.

Keywords: Distributed Evolutionary Algorithms, Cloud Computing

1 Introduction

Information technology has become ubiquitous in today's world, sources of computing power range from personal computers and smart-devices to massive data centers. Users can now access vast computational resources available on the Internet using diverse technologies, including cloud computing, peer-to-peer (P2P), and http-based environments. This trend can favor Evolutionary Computation (EC) algorithms as these can be designed as parallel, distributed, and asynchronous systems. Several Evolutionary Algorithms (EA) have been proposed that distribute the evolutionary process among heterogeneous devices, not only among controlled nodes in a in-house cluster or grid but also in those out side the data center, in users' web browsers and smart phones or external cloud based virtual machines. This reach out approach allows researchers the use of low cost computational power that would not be available otherwise, but on the other hand, have the challenge to manage heterogeneous unreliable computing

resources. Lost connections, low bandwidth communications, abandoned work, security and privacy issues are all common in these settings.

In this paper the effect of node unavailability in algorithms using the EvoSpace population storage is assessed. EvoSpace [*references hidden for blind review*] is a framework to develop evolutionary algorithms (EA) using heterogeneous and unreliable resources. EvoSpace is based on Linda's tuple space [8] coordination model, where each node asynchronously pulls its work from a central shared memory. The core elements of EvoSpace are a central repository for the evolving population and remote clients here called EvoWorkers which pull random samples of the population to perform on them the basic evolutionary processes (selection, variation and survival), once the work is done, the modified sample is pushed back to the central population. This model contrasts with the use of a global queue of tasks and implementations of map-reduce algorithms, recently favored in other proposals [5, 4, 11]. Following the tuple space model, when individuals are pulled from the EvoSpace container these are removed from it, so that no other EvoWorker could work on them at same time. This design decision has several known benefits relevant to concurrency control in distributed systems, and also is an effective way of distributing the workload. Leaving a copy of the individual in the population server free to be pulled by other EvoWorkers will result in redundant work and this could be costly if the task at hand is time consuming. EvoWorkers are expected to be unreliable, as they can loose a connection or are simply shut down or removed from the client. When an EvoWorker is lost, so are the individuals pulled from the repository. Depending on the type of algorithm been executed, the lost of these samples could have a high cost. To address the problem of unreliable EvoWorkers, EvoSpace uses a simple re-insertion algorithm that also prevents the starvation of the population pool. Other pool based algorithms normally use a random insertion technique, but we argue this could negatively impact the outcome of the algorithm in some cases.

This work evaluates the effect of the re-insertion algorithm has on the total running time and number of evaluations of a genetic algorithm. Using a benchmark problem from the P-Peaks problem generator, we compare both approaches: (i) the re-insertion previous individuals at the cost of keeping copies of samples, and (ii) inserting randomly generated individuals, with the sometimes beneficial cost of adding diversity to the population. For this experiments we use the same parameters used in an earlier work, in order to compare the performance of the algorithm in similar conditions. EvoSpace was implemented as a web service on the popular Heroku platform and EvoWorkers where simulated using PiCloud, a scientific computing PaaS.

The remainder of the paper proceeds as follows. Section 2 reviews related work. Afterwards, Section 3 briefly describes the proposed EvoSpace framework and gives implementation details the re-insertion process. The experimental work is presented in Section 4. Finally, a summary and concluding remarks are in Section 5.

2 Related Work

Using available Internet resources for EC has been the focus of recent research in the field. The use of volunteer computing using BOINC open source software is used by Smaoui et al. [6] in this case BOINC uses redundancy to deal with the volatility of nodes and unreliability of their results. In this work each BOINC work unit consisted of a fitness evaluation task and multiple replicas were produced and sent to different clients, later when outputs were received a validation step ensured all outputs match. In case of a discrepancy or a time-out from a client, a new job replica was created and sent to another client. The main drawback of this approach was that the master-worker algorithm used was synchronous, so the process had to wait for all jobs to continue to the next generation. Web browsers were used by Merelo et al. [10] using Javascript to implement the algorithm, this has the advantage of not requiring the installation of additional software. In this work the server receives an Ajax request with the best individual obtained from the local evolution in clients, and then responds with additional parameters and the best individual in the population so far. If a client is disconnected no special measures are taken. Several cloud-based EC solutions are based on a global queue of tasks and a Map-Reduce implementation which normally handles failures by the re-execution of tasks [5, 4, 11].

3 EvoSpace

EvoSpace [*references hidden for blind review*] consists of two main components (see figure 1): (i) the EvoSpace container that stores the evolving population and (ii) EvoWorkers, which execute the actual evolutionary process, while EvoSpace acts only as a population repository. In a basic configuration, EvoWorkers pull a small random subset of the population, and use it as the initial population for a local EA executed on the client machine. Afterwards, the evolved population from each EvoWorker is returned to the EvoSpace container. When individuals are pulled from the container they remain in a phantom state, they cannot be pulled again but they are not deleted. Only if and when the EvoWorker returns the replacement sample phantoms are truly deleted. If the EvoSpace container is at risk of starvation or optionally when a time-out occurs new phantom individuals are re-inserted to the population and available again. This can be done because a copy of each sample is stored in a priority queue used by EvoSpace to re-insert the sample to the central population; similar to games where characters are respawned after a certain time. In the experiments conducted in this work re-insertion occurs when the population size is below a certain threshold. Figure 1 illustrates the main components of EvoSpace.

3.1 Implementation

Populations of individuals are stored in-memory, using Redis, a key-value database, which was chosen over a relational database system, or other non-SQL alternatives, because it provides a hash based implementation of sets and queues which

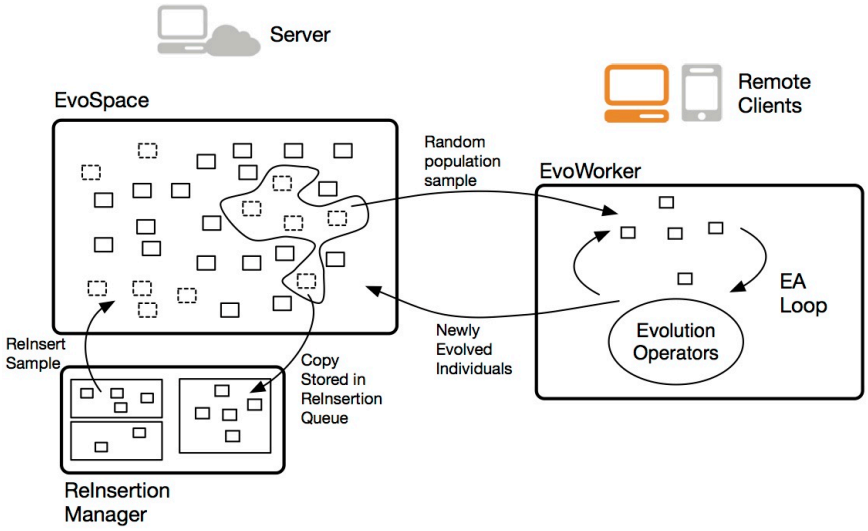

Fig. 1. Main components and dataflow within EvoSpace.

are natural data structures for the EvoSpace model. The logic of EvoSpace is implemented as a python module and exposed as a web service using Cherrypy http library. The EvoSpace modules are available with a Simplified BSD License from `anonymousURL`.

3.2 Evospace as a Heroku Application

Heroku (`http://heroku.com`) is a multi-language PaaS, supporting among others Ruby, Python and Java applications. The basic unit of composition on Heroku is a lightweight container running a single user-specified process. These containers, which they call *dynos*, can include web (only these can receive `http` requests) and worker processes (including systems used for database and queuing, for instance). These process types are the prototypes from which one or more dynos can be instantiated; if the number of requests to the server increases more instances can be assigned on-the-fly. In our case, our CherryPy web application server runs in one web process, when the number of workers was increased we added more dynos (instances) of the CherryPy process.

This model is very different from a VPS where users pay for the whole server; in a process based model, users pay only for the processes they need; being a *freemium* model means also that, if a minimum level of resources is not exceeded, it can be used for free.

Once deployed the web process can be scaled up by assigning more dynos; in our case and in the more demanding configurations of our experiments, the web process was scaled to 20 dynos. Instructions and code for deployment is available at `anonymousURL`

3.3 Evoworkers as PiCloud Jobs

PiCloud is a Platform as a Service (PaaS), with deep Python integration; using a library, Python functions are transparently uploaded to PiCLoud's servers as units of computational work they call *jobs*. Each job is added to a queue, and when there is a core available, the job is assigned to it. Both Heroku and PiCloud platforms are deployed on top of Amazon Web Services (AWS) infrastructure in the US-EAST Region. This ensures minimal latency and a high bandwidth communication between the services, and there is no charge for data transfer costs between both services. For the experiments type c1 and c2 Real Time workers where used. The code for the EvoWorkers implementation and experiment data is publicly available from a github repository `anonymousURL`.

4 Experimental work

4.1 Benchmark

The experiment reported here uses a multimodal problem generator. A P-Peaks generator has been chosen because the problem (and the computing resources needed for the search) can be appropriately scaled. Proposed by De Jong et al. in [2] a P-Peaks instance is created by generating a set of P random N-bit strings, which represent the location of the P peaks in the space. To evaluate an arbitrary bit string $\mathbf{x}$ first locate the nearest peak (in Hamming space). Then the fitness of the bit string is the number of bits the string has in common with that nearest peak, divided by N. The optimum fitness for an individual is 1. This particular problem generator is a generalization of the P-peak problems introduced in [3].

$$f_{P-PEAKS}(\mathbf{x}) = \frac{1}{N} \max_{i=1}^{P} \{N - hamming(\mathbf{x}, Peak_i)\} \tag{1}$$

A large number of peaks induce a time-consuming algorithm, since evaluating every string is computationally hard; this is convenient since to evaluate these type of distributed evolutionary algorithms fitness computation has to be significant with respect to network latency (otherwise, it would always be faster to have a single-processor version). However according to Kennedy and Spears [9] the length of the string being optimized has a greater effect in determining how easy or hard is the problem. In their experiments an instance having P = 200 peaks and N = 100 bits per string is considered to produce a considerably difficult problem.

4.2 Experimental Set-up

As EvoSpace is only the population store, EvoWorkers must implement the genetic operators. The genetic algorithm executed by EvoWorkers has been implemented using a modified DEAP (Distributed Evolutionary Algorithms in Python) framework [7] GA. Is important to note that only the basic non-distributed GA library was used. Three methods were added to the local algorithm: `getSample()`

Table 1. GA and EvoWorker parameters for experiments.

GA Parameters	
Tournament size	4
Crossover rate	0.85
Population Size	512
Mutation probability	0.5
Independent bit flip probability	0.02
EvoWorker Parameters	
Sample Size	16
Generations	128
Other Parameters	
PiCloud Worker Type	Realtime
Number of Workers	4,8,16
Return Sample Probability	30%,50%,90%
Number of Executions	30

and `putBack()`; and another for the initialization of the population. The implementation of the local GA simply uses DEAPs methods; for instance to generate the initial population, a local `initialize()` is called and the population sent to EvoSpace.

The selection of parameters was based on those used in [1]: a tournament size of 4 individuals, a crossover rate of 0.85 and a population of 512 individuals. In [2] a mutation rate equal to the reciprocal of the chromosome length; is recommended, as DEAP uses two parameters they were defined as follows, mutation probability of 0.5 and an independent flip probability of 0.02. For EvoWorkers the parameters were 128 worker generations for each sample, and a sample size of 16. To simulate unreliable workers each worker was assigned a return sample probability. In the experiments the lower probability was a 30% chance of an EvoWorker returning a sample or an EvoWorker failing 70% of the time; other return sample probabilities where 50%, 70% and 90%. Experiments where carried out for 4, 8 and 16 EvoWorkers. In a pool based asynchronous GAs there is usually no need to wait for a workers job to start a new generation. Although supported by EvoSpace time outs were not chosen as triggers to feed the population with new individuals, the population size was used instead. We believe the population size is a better threshold as it is more critical to the GA performance. In a previous work we found that when the population remaining in the pool was near starvation, the time of completion was increased. For these experiments, the insertion of individuals was triggered when less than 128 individuals remain in the population; the number of individuals feed to the population was 128, or 8 samples when the re-insertion algorithm was used. A summary of the setup is presented in table 1.

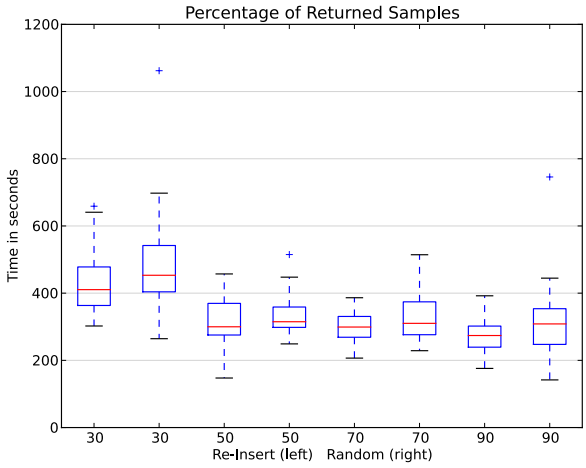

Fig. 2. Time required to solution, 4 Workers

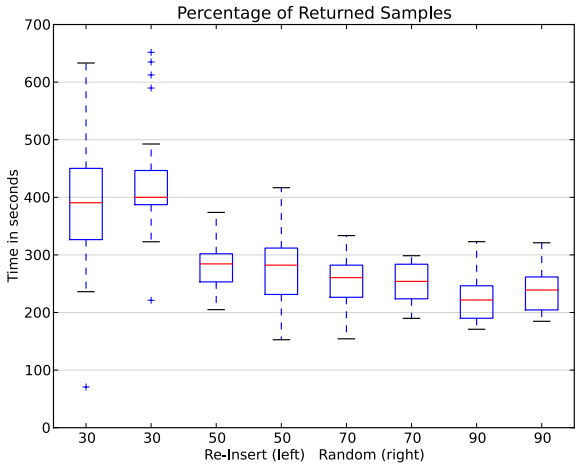

Fig. 3. Time required to solution, 8 Workers

4.3 Results

In this section, results from the experiments are discussed. Figure 2 shows the time required to solution when using four EvoWorkers. For a population of 512 individuals and a sample size of 16, there is no difference in the time required to solution for percentages of 50% and above. Both re-insertion algorithms had comparable times. For 30 percent, both approaches had slight increase in time. For 8 workers (see Figure 3) there was marginal decrease in overall time; and

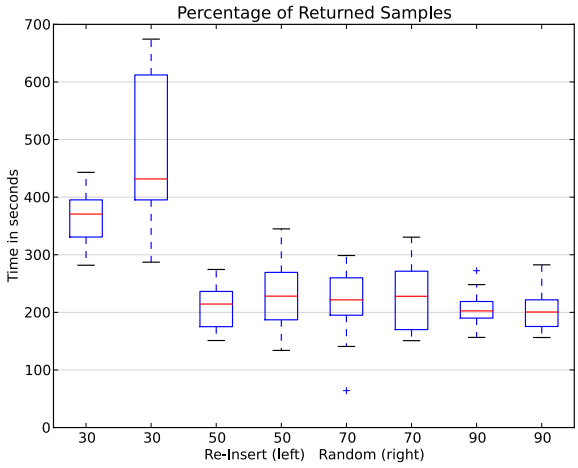

Fig. 4. Time required to solution, 16 Workers

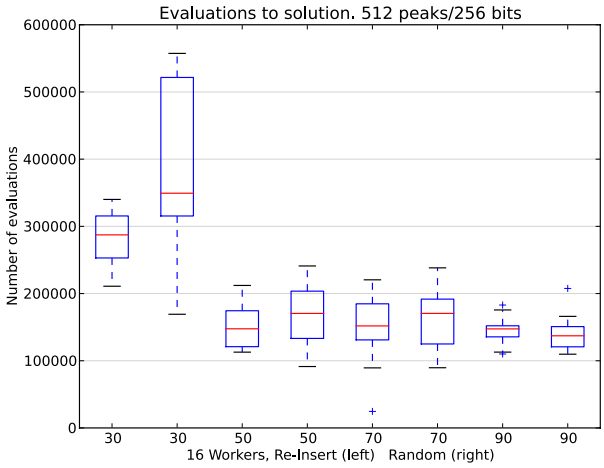

Fig. 5. Number of evaluations needed to solution, 16 Workers

results where similar to those found in the experiments with 4 workers. Figure 4 shows results for 16 workers, when there was only a 30% chance of returning a sample the rate of re-insertions was high, approximately once every 35 samples. In this case, the insertion of random individuals resulted in a higher time to solution. For these experiments when there is not a high rate of re-insertion, both alternatives have similar results, but the re-insertion algorithm is better for situations when there are many starvation conditions. It appears that the

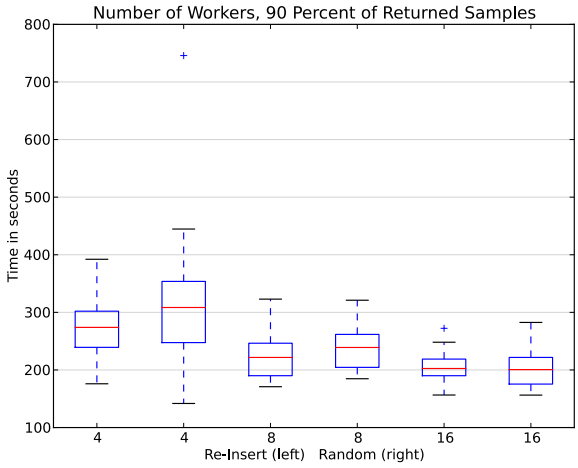

Fig. 6. Time required to solution, 90% of returned samples

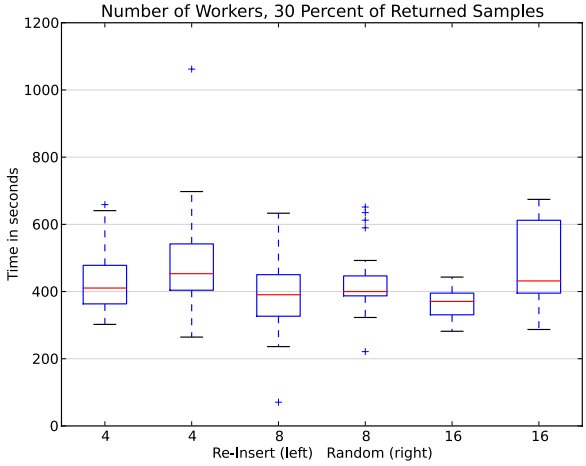

Fig. 7. Time required to solution, 30% of returned samples

insertion of random individuals is not detrimental when there are other evolved individuals in the pool. But when the remaining pool almost consists of random individuals, samples pulled by EvoWorkers need to start the search all over again. If not many samples are then returned to the pool, the work needed to reach an optimum is increased. Figure 5 also shows the number of evaluations needed to reach an optimum again for 16 workers. Figures 6 and 7 show the time required to solution for 30 and 90 percent of returned samples. For 90% both algorithms

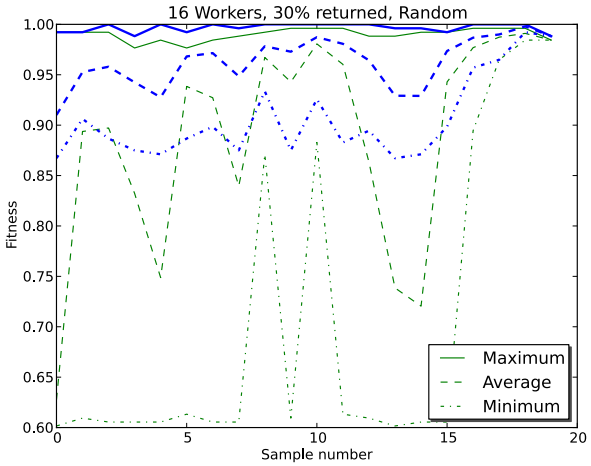

Fig. 8. Fitness by Sample number, 30% of returned samples, 16 Workers, Random Algorithm

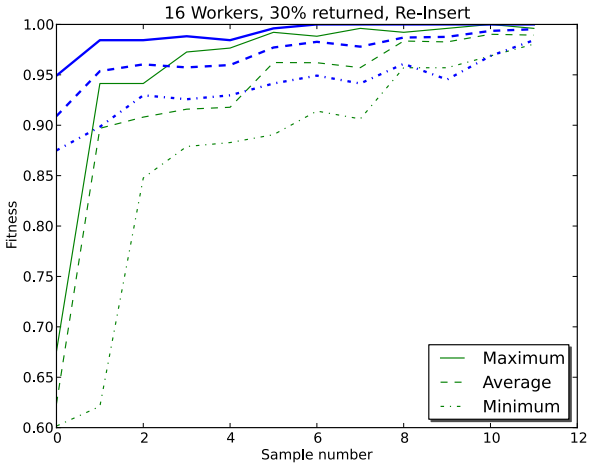

Fig. 9. Fitness by Sample number, 30% of returned samples, 16 Workers, Re-Insert Algorithm

had similar speedups when incrementing the number of workers. The marginal speedup obtained for these experiments is related to the population size, but this parameter was not changed. For 30% there was practically no speedup at all. The re-insert algorithm although not significant, had consistent decrease in time.

Fitness by time was measured as the average from each consecutive sample pulled by each worker. For each sample the average fitness was measured at the start and at the end of the local evolution. Also the minimum and maximum fitness values at start and finish was recorded. Final fitness values are shown in double width lines in figures. Readings used for these figures include all samples, including those that where not returned. Figure 8 shows fitness by time for the random insertion algorithm, as expected initial fitness drops at certain points, when random insertion occurs. Average final fitness is affected by random insertions.Figure 9 shows results for the re-insertion algorithm, with more characteristic curves for this type of algorithms.

5 Conclusions and Further Work

The re-insertion algorithm proposed for EvoSpace is a viable alternative to deal with a starving population in pool based GAs, and unreliable EvoWorkers. Using a benchmark problem from the P-Peaks problem generator, the approach was compared against the option of inserting randomly generated individuals. The same parameters and computing resources were used when testing both algorithms with better times reported when using the proposed technique. For experiments where the population size is enough for the number of workers with their sample sizes, plus a buffer to account for loss samples, then both algorithms could be used. However for cases when the number of EvoWorkers is unknown an hybrid approach could be used, insertion of random individuals to gradually increase the population size, and a re-insertion queue to handle lost samples. Further work could be focused on a hybrid approach, and also using heterogeneous computing resources.

6 Acknowledgements

Hidden for double-blind review.

References

1. E. Alba, A. J. Nebro, and J. M. Troya. Heterogeneous Computing and Parallel Genetic Algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362–1385, Sept. 2002.
2. K. A. De Jong, M. A. Potter, and W. M. Spears. Using problem generators to explore the effects of epistasis. In T. Bck, editor, *ICGA*, pages 338–345. Morgan Kaufmann, 1997.
3. K. A. De Jong and W. M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, pages 38–47, London, UK, UK, 1991. Springer-Verlag.

4. S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro. Towards migrating genetic algorithms for test data generation to the cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline.*, pages 113–135. IGI Global, IGI Global, 2013.
5. P. Fazenda, J. McDermott, and U.-M. OReilly. A library to run evolutionary algorithms in the cloud using mapreduce. *Applications of Evolutionary Computation*, pages 416–425, 2012.
6. M. S. Feki, V. H. Nguyen, and M. Garbey. Parallel genetic algorithm implementation for boinc. In B. M. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. J. Peters, and T. Priol, editors, *PARCO*, volume 19 of *Advances in Parallel Computing*, pages 212–219. IOS Press, 2009.
7. F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.
8. M. García-Valdez, L. Trujillo, F. Fernández de Vega, J. J. Merelo Guervós, and G. Olague. EvoSpace: A Distributed Evolutionary Platform Based on the Tuple Space Model. In A. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 499–508. Springer Berlin Heidelberg, 2013.
9. J. Kennedy and W. Spears. Matching algorithms to problems: an experimental test of the particle swarm and some genetic algorithms on the multimodal problem generator. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 78–83, May.
10. J. Merelo-Guervos, P. Castillo, J. L. J. Laredo, A. Mora Garcia, and A. Prieto. Asynchronous distributed genetic algorithms with Javascript and JSON. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 1372–1379, June.
11. D. Sherry, K. Veeramachaneni, J. McDermott, and U.-M. OReilly. Flex-gp: Genetic programming on the cloud. In C. Chio, A. Agapitos, S. Cagnoni, C. Cotta, F. Vega, G. Caro, R. Drechsler, A. Ekrt, A. Esparcia-Alczar, M. Farooq, W. Langdon, J. Merelo-Guervs, M. Preuss, H. Richter, S. Silva, A. Simes, G. Squillero, E. Tarantino, A. Tettamanzi, J. Togelius, N. Urquhart, A. Uyar, and G. Yannakakis, editors, *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 477–486. Springer Berlin Heidelberg, 2012.